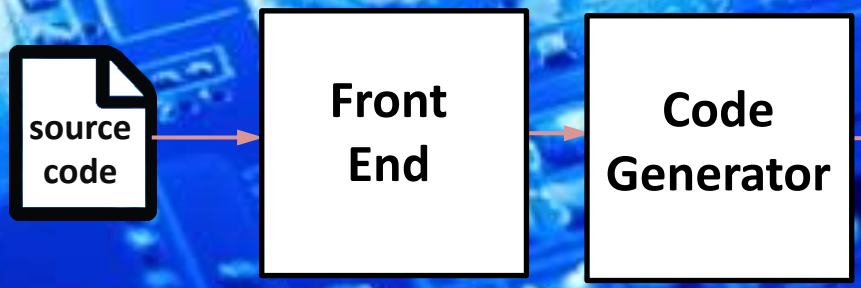
Embedded Systems II



GNU Debug (GDB) & Bash

Towards Prac1 'prac lecture'

Lecture P1

Embedded Systems II

Dr Simon Winberg

Electrical Engineering
University of Cape Town

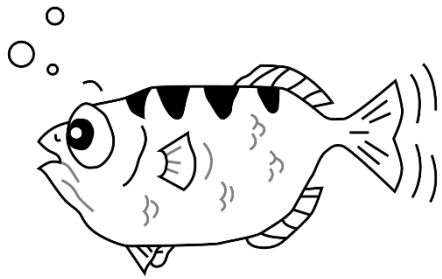
Overview

- Introducing Gnu DeBug GDB
- Using GDB from the command line
- Simple example of using GDB
- Remote GDB
- The Born Again SHell (BASH)

Comments: this lecture was planned more around an introduction to prac1 as apposed to a more theory lecture. These things aren't going to be examined, with the possible exception of slide 4 which discusses why you might need a debugger.

Introducing
GDB
The Gnu DeBugger

Homepage of the GDB project:
https://www.gnu.org/software/gdb/



Why GDB?

- Many reasons you may need a debugger…
- *The main reason:* you've written a program that doesn't work quite right.
- Other common reasons: open-source depends on sharing source code and on porting code from other systems such as Linux, Windows, Mac, etc… which might work fine on the system it was originally developed for but may have problems when you try running it on your platform… sifting through thousands of lines of code to find potential problems is… for sure not fun ☹ and could take ages.
- *Therefore* it's worth figuring out a good C debugger, and especially one that has wide support and a large community of users… GDB probably top of this list!

GDB Basics

- GDB was originally developed by the Free Software Foundation, and works with both C and C++ code (various other languages are also suppored, e.g. FORTRAN and the list is growing).
- GDB lets you
  - Stop / Start execution of a program
  - Examine and change variables during execution
  - Trace how the program executes (remember the call stack … it lets you inspect this).
  - It also has command line editing and history features similar to a regular console, such as bash or DOS.
  - A graphical interface has been added as well… e.g.:
    - Gede : http://gede.acidron.com/ (smaller footprint, OK; not used recently)
    - Although usually I don't use GDB outside an IDE
  - Then lots of IDEs with GDB integrated
    - Code::Blocks : http://www.codeblocks.org/ (my current favorite!)
    - Kdevelop : https://www.kdevelop.org/ (used to love this one!)
    - Cloin: https://www.jetbrains.com/clion/ (very high quality, but a bit commercial)
- But… sometimes you don't have much space and only a console to work with, and in that case, the basic GDB console interface is what you're stuck with (unless you have an expensive ICE module where you can do the debugging remotely on a PC with the appropriate fancy software).

GDB Manual

- The FSF has a comprehensive manual to explain how to use GDB, see:

  https://sourceware.org/gdb/onlinedocs/gdb/

- But I'll summarize some of the essentials …

Preparing your executable for GDB

- Regular compiler settings are not going to include debugging data (e.g. symbol names, addresses, etc.) in the executable. You can still run GDB on these files, but it's going to be super hard to use, as you'll have variables names things like 0xFAAA001 and the like according to addresses they are assigned. The solution is to set GCC to include debugging data into your executable.
- Use the -g option when compiling and linking your code with gcc to include debug data, e.g.:

```
$ gcc -g main.c util.c
```

Tip re compiler optimization: if your program isn't working with optimization flags, remove the optimization. If still doesn't work without optimization, then apply GDB to that unoptimized code. It is best to use gdb on unoptimized code, because optimized code diverges more from the original source code you're trying to fix.

Invoking GDB on your program

- To invoke GDB simply use:

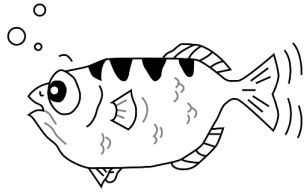
  $ gdb program [core-dump]

- Where *program* is the filename of the executable file to debug. You can optionally include core-dump, the name of a core dump file left from an earlier execution attempt, that you can use to inspect where your program crashed.
- The GDB startup messages are fairly verbose, telling you the gdb version used, how the core file was generated (which program and in what function), among other details.
- From this point you can start entering GDB commands. Use ‘quit’ or Ctrl-D to exit.

Simple Intro to

command line GDB

Probably want GDB manual open to see commands to use:
https://www.sourceware.org/gdb/documentation/



but first we need a faulty program to test GDB with …

Example C program to test with

```
/**  Simple program to illustrate how GDB works.
     This program should generate a segfault at line 25.*/

/** Include libraries */
#include <stdio.h>
#include <string.h>

/** A looping function */
void copyend (int n, char* a, char* s)  {
  int i;
  for (i=0; i<n; i++) strcat(s,a);
}

/** Entry point to the program, which calls some functions */
int main() {
   char s[10] = "Hello";  /* takes 6 bytes of 10 available */
   char t[10] = "     ";  /* a temporary string */
   printf("Testing program started!\n");
   printf("Start: s='%s'  t='%s'\n",s,t);
   copyend(2,"a",s);  /* copy string "a" to end of s 2x - should be fine  */
   copyend(3,NULL,s); /* copy null to end of s 3 times - causes a problem */
   printf("End  : s='%s'  t='%s'\n",s,t);
   printf("Testing program ended without crashing!\n");
   return 0;
}
```

A super simple makefile (demonstrating macros)

```
CC=gcc
# note the -g below for debugging
CFLAGS=-I. -g
DEPS = test1.h
OBJ = test1.o

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

test1: $(OBJ)
	$(CC) -o $@ $^ $(CFLAGS)

clean:
	rm -f -r test1 *.o
```

Example run

```
$ make
gcc -c -o test1.o test1.c -I.
gcc -o test1 test1.o -I.
$ ./test1
Testing program started!
Start: s='Hello'  t='     '
Segmentation fault
```

Let's start up gdb to see where it segfaults

GDB Help and useful command sets (in red)

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in
that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

… but see the next slides for simple use cases!

GDB – can just running the program and see where it crashes…

```
$ gdb ./test1
(gdb) run
Starting program: test1
Testing program started!
Start: s='Hello'  t='     ‘

Program received signal SIGSEGV, Segmentation fault.
__strcat_sse2 () at ../sysdeps/x86_64/multiarch/../strcat.S:166
166     ../sysdeps/x86_64/multiarch/../strcat.S: No such file or directory.
```

(I’ve obviously cut out all the start-up text gdb outputs)

- So in this case, we simply just said run and relied on gdb to trap whatever was causing the segfault.
- This isn’t necessary the most useful result, as the string library (and strcat in particular) was not combined with -g enabled (we were using optimized, pre-compiled libraries). You don’t really want to go a recompile the standard libraries to support debugging, those are generally fine and you don’t want to delve into them. But we know the problem happened in a call to strcat.
- → Next step: so we want to find out where in the program we got to. This can be done using the **bt** command in gdb… (see next slide)

To see where the crash happened, do this…

The backtrace or **bt** command prints out the stack frames, representing this data usefully, showing the address in memory where the program was stopped, the parameters passed, and the filename of the c module where the function was implemented. As well as the various function calls leading up to this one.

```
(gdb) bt
#0  __strcat_sse2 () at ../sysdeps/x86_64/multiarch/../strcat.S:166
#1  0x000000000040065c in copyend (n=3, a=0x0, s=0x7fffffffe4e0 "Helloaa")
      at test1.c:13
#2  0x00000000004006f9 in main () at test1.c:25
```

To see active variables

These are useful commands for displaying variables:

info variables : lists all global and static variable names (masses, not so useful!)

info locals : to list all local variables in current stack frame (names, values) and includes any static variables in that function.

info args : list arguments of the current stack frame (names, values)

Note, if you've got to a segfault, you probably want to move back to a previous stack frame and query the variables there, otherwise you'll get a response like:

(gdb) **info locals**
No locals.

(see next slide how to move around the stack fames)

Moving around stack frames

This is useful for inspecting local variables in functions that were called in leading up to a segfault.

First you want to know what stack frames are available. Simply use **bt** again.

```
(gdb) bt
#0  __strcat_sse2 () at ../sysdeps/x86_64/multiarch/../strcat.S:166
#1  0x000000000040065c in copyend (n=3, a=0x0, s=0x7fffffffe4e0 "Helloaa")
      at test1.c:13
#2  0x00000000004006f9 in main () at test1.c:25
```

We are currently in stack frame #0, which for which we have no access to the symbol table and don't know what the local variables are because we do not have the debugging information for these library implementations.
Thus, let's move back a frame to #1, which is done as follows:

(gdb) **select-frame 1** (you can abbreviated this as: **sel 1**)

And now you can view the local variables:

(gdb) **info locals** *i.e. list local variables*

i = 0

Or view the parameters using:

(gdb) **info args**

n = 3

a = 0x0

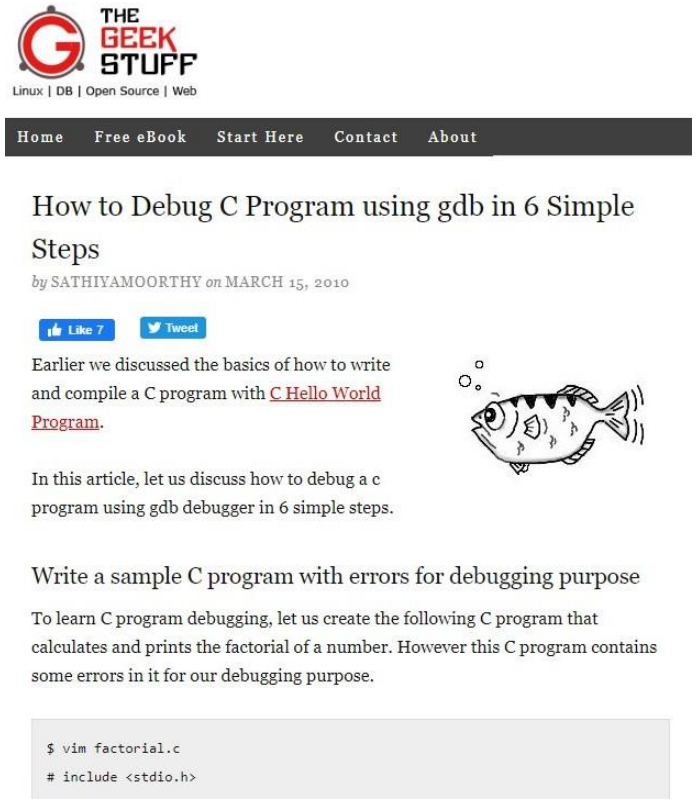
s = 0x7fffffffe4e0 "Helloaa"

That should ring a warning bell for you, the programmer, as it should be a valid string!

Becoming familiar with GDB

I think the best way to learn about GDB is to **try it out for yourself**. I don't really think a YouTube is all that useful in this case, particularly for using GDB from the console. I do strongly recommend working though this very convenient tutorial, you can use GDB either on a regular PC (e.g. under Ubuntu) or the Pi (this tutorial is relevant to both options).

https://www.thegeekstuff.com/2010/03/debug-c-program-using-gdb/



PS: And along with the suggested tutorial, you might want to explore TheGeekStuff further, it's got masses of tips and techniques, that can help you become a Linux guru.
See: https://www.thegeekstuff.com/

Great, now we now know about running GDB at least on a PC or logging into a Linux console to use it… but is running GDB locally sufficient for an embedded systems engineer to know about?!

… to answer this: **not really**, you should also be aware of:

Embedded Developers should know of

Remote GDB

…

Remote GDB

- The Linux kernel implements the ‘ptrace’ system calls, which means you don’t really need to add ‘gdb stubs’* to debug embedded applications remotely.
- Instead, a gdb server is provided with the gdb installation package for Linux (i.e. using apt-get).
- This gdb server is a small program (daemon) that runs on the target and executes the commands it receives from the gdb debugger running on the host.
- Hence, any application can be debugged on the target without having the gdb debugger actually running on the target. This is important to know: because

  (a) The gdb binary is big (for a memory-limited platform)

  (b) It can substitute for many of the features only available from a costly ICE device.

* gdb stubs are hooks and handlers placed in the target firmware or its operating system kernel to allow interaction with a remote debugger. The gdb manual explains the use of gdb stubs. If you weren’t using Linux, we’d be talking about using GDB stubs; good to know about in case you need to use a different OS for your embedded system.

Remote GDB – Howto

- It's a bit of a process involved in getting remote GDB set up. It isn't so necessary for the Raspberry Pi anyway, as it has lots of memory and can run a full IDE with GUI.
- Only if you have a very constrained embedded platform or your system is in a difficult position or place to access, would you need to consider using the GDB server.
- There are a number of tutorials on the web that explain how to do it. Among the easiest (free or low-cost) options is to consider JetBrains solutions:

  https://www.jetbrains.com/help/clion/embedded-gdb-server.html

Introducing
Bash
(the '**B**ourne **A**gain **SH**ell')

Homepage of the GNU Bash project:
https://www.gnu.org/software/bash

What is Bash?

- Bash is GNU's **shell** and scripting program.
- BASH is technically an acronym that stands for the 'Bourne Again Shell' (the GNU site explains the story behind the name) but one major aspect is it is compatible with the older *sh* shell, *ksh* (Korn shell) and C shell (csh).
- It offers functional improvements over these other shells, both for programming and for interactive uses.
- A **shell** in computing is a command line text interface to the operating system.

Using Bash

- Best way to learn Bash (like many practical computing applications) is to use it.
- Linux (most popular flavours at least, like Ubuntu, RedHat and indeed Raspbian) use Bash by default – when you open a terminal in Raspbian it gives you a Bash console.

Using Bash

This is a list of very commonly used Bash command, to get you started

Command	Description
ls	List current files and folders in directory
ls -al	List details about everything
pwd	Print current (working) directory
cd <directory>	Move into the specified directory
cd ..	Move up one directory
ifconfig	Shows details on current network interfaces (requires package net-tools to be installed)
touch <file>	Creates a file. For example "touch.txt" will create a file in the directory you are in.
nano <file>	Opens the specified file in nano, a text editor
mkdir <directory>	Creates a directory
sudo <command>	Executes <command> with super user privilages
man <command>	Opens the user manual relating to the specified command

Using Bash

- The first prac in this course encourages you to do some experimenting with Bash
- In later pracs, and indeed later in your career, you may want to leverage Bash for scripting and automation purposes, here's a suggested site to look at if you want to get into that:
  - This HowToForge page gives a good introduction to using and writing Bash scripts: https://www.howtoforge.com/tutorial/linux-shell-scripting-lessons/
  - This site provides an intro to Bash as well as a free e-book (Open Commons) that gets into details of advanced programming with Bash: https://opensource.com/article/20/4/bash-programming-guide

End of Lecture

The Next Episode…

Lecture L04

Next lecture:
Selection Process, Golden Measures,
Benchmarking & Code Reviews